Debugging with dbx

Bill Tuthill

Sun Microsystems, Inc.
2550 Garcia Avenue

Kevin J. Dunlap

Computer Systems Research Group
University of California
Berkeley, CA 94720

Introduction

This short paper discusses *dbx*, a symbolic debugger that is vastly superior to *adb*. It may be as good as the debuggers you remember from those non- UNIX† systems you worked on before. The advantage of symbolic debuggers is that they allow you to work with the same names (symbols) as in your source code.

Like *adb*, *dbx* is interactive and line-oriented, but *dbx* is a source-level rather than an assembly-level debugger. It allows you to determine where a program crashed, to view the values of variables and expressions, to set breakpoints in the code, and to run and trace a program. Source code may be in C, Fortran, or Pascal.

Mark Linton wrote *dbx* as his master's thesis at UC Berkeley. Along with Eric Schmidt's Berknet, *dbx* is among the most successful master's theses done on UNIX. Since *dbx* required changes to the symbol tables generated by the various compilers, you need to compile programs for debugging with the *−g* flag. For example, C programs should be compiled as follows:

% cc −g *program*.c −o *program*

Programs compiled with the *−g* option have good symbol tables, while programs compiled without *−g* have old-style symbol tables intended for *adb*. Stripped programs have no symbol tables at all. Invoke the debugger as follows, where *program* is the pathname of the executable file that dumped core:

% dbx *program*

The core image should be in the working directory; if it isn't, specify its pathname in the argument after the program name. Among the great advances of *dbx* is that it has a help facility; type the *help* request to see a list of possible requests. You can obtain help on any *dbx* request by giving its name as an argument to *help*.

† UNIX is a Trademark of Bell Laboratories.

Examining Core Dumps

Much of the time, programmers use *dbx* to find out why a program dumped core. As an example, consider the following program *dumpcore.c*, which dereferences a NULL pointer. This is a legal operation on VAX/UNIX, but not on VAX/VMS or on MC68000-based UNIX systems, on one of which this example was run:

```
#include <stdio.h>

#define LIMIT 5

main()                      /* print messages and die */
{
        int i;

        for (i = 1; i <= 10 ; i++) {
                printf("Goodbye world! (%d)\n", i);
                dumpcore(i);
        }
        exit(0);
}

int *ip;

dumpcore(lim)               /* dereference NULL pointer */
int lim;
{
        if (lim >= LIMIT)
                *ip = lim;
}
```

The program core dumps because of a segmentation violation or memory fault — on most machines it is illegal to assign to address zero. Once the program has produced a core dump, here's how you can find out why the program died:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) where
dumpcore.dumpcore(lim = 5), line 22 in "dumpcore.c"
main(0x1, 0x7fffe904, 0x7fffe90c), line 11 in "dumpcore.c"
```

The *where* request yields a stack trace. As you can see, the *dumpcore()* routine was called from line 11 of the program, with the argument *lim* equal to 5. You can look at the *dumpcore()* procedure by invoking the *list* request as follows:

```
(dbx) list dumpcore
   18   dumpcore(lim)          /* dereference NULL pointer */
   19   int lim;
   20   {
   21           if (lim >= LIMIT)
   22                   *ip = lim;
   23   }
```

We immediately suspect that the program's failure had something to do with **ip*, so we use the *print* request to retrieve the value of the pointer and what it points to:

```
(dbx) print *ip
reference through nil pointer
(dbx) print ip
(nil)
```

This tells us the program has dereferenced a null pointer. It is possible to run the program again from inside the debugger. The first line tells you name of the running program, and successive lines give output from the program:

```
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
Goodbye world! (5)

Bus error in dumpcore.dumpcore at line 22
   22          *ip = lim;
(dbx) quit
```

In this example the program dies with a Bus error at line 22. This method of running the program does not produce a core dump, but the *where* request will still behave properly, because the debugger is in the same state as if it had just read the core file.

Setting Breakpoints

With *dbx* you can set breakpoints before each line of a program, not just at function and procedure boundaries, as with *adb*. The *stop* request sets a breakpoint. After setting a breakpoint, use the *run* request to execute the program. The *cont* request continues execution from the current stopping point until the program finishes or another breakpoint is encountered. The *step* request executes one source statement, following any function calls. The *next* request executes one source statement, but does not stop inside any function calls. The *status* request lists active breakpoints, while the *delete* request removes them if required.

The *stop* request can take a conditional expression to avoid needless single-stepping. We will use a conditional in our example to make things simpler. Of course you can use *print* and *list* requests at any time during statement stepping if you want to print the value of variables or list lines of source code. This sample session shows a mixture of requests as we verify that the program fails when it tries to assign to **ip*:

```
(dbx) stop at 10 if (i == 5)
[1] if i = 5 { stop } at 10
(dbx) run
Goodbye world! (1)
Goodbye world! (2)
Goodbye world! (3)
Goodbye world! (4)
[1] stopped in main at line 10
   10          printf("Goodbye world! (%d)\n", i);
(dbx) next
Goodbye world! (5)
stopped in main at line 11
   11          dumpcore(i);
(dbx) step
stopped in dumpcore at line 21
   21      if (lim >= LIMIT)
(dbx) step
stopped in dumpcore at line 22
   22          *ip = lim;
(dbx) step
Bus error in dumpcore.dumpcore at line 22
   22            *ip = lim;
```

Running the program with breakpoints assures us that our intuition was correct. We shouldn't be assigning anything to a null pointer — *ip* should have been initialized to point at an object of the proper type. To exit from the debugger, use the *quit* request.

It is possible to set variables from inside *dbx*. The previous breakpoint session, for example, could have gone like this:

```
% dbx dumpcore
dbx version 3.17 of 4/24/86 15:04 (monet.Berkeley.EDU).
Type 'help' for help.
reading symbolic information ...
[using memory image in core]
(dbx) stop at 10
[1] stop at 10
(dbx) run
Running: dumpcore
stopped in main at line 10
   10          printf("Goodbye world! (%d)\n", i);
(dbx) assign i = 5
(dbx) next
Goodbye world! (5)
stopped in main at line 11
   11          dumpcore(i);
(dbx) next
Bus error in dumpcore.dumpcore at line 22
   22            *ip = lim;
```

It is often useful to assign new values to variables to draw conclusions about alternative conditions. We can't fix the bug in this program, however, because there is no declared variable to which *ip* should point.

Conclusion

Expressions in *dbx* are similar to those in C, except that there is a distinction between / (floating-point division) and *div* (integer division), as in Pascal. The table on the following page shows *dbx* requests organized by function:

Like *adb*, *dbx* can disassemble object code. It can also examine object files and print output in various formats; but *dbx* requires the proper symbol tables, so *adb* is more useful to examine arbitrary binary files. The most important thing *adb* can do that *dbx* cannot is to patch binary files — *dbx* has no write option. Despite these shortcomings, *dbx* is much easier to use than *adb*, so it contributes much more to individual programmer productivity.

Acknowledgements

Material presented in this document was first presented in ‘‘C Advisor’’, *Unix Review 4,* 1, pp 78–85. The Regents of the University California expresses their gratitude to Unix Review for allowing them to reprint this document.

This document is a good starting point for a more thorough tutorial. Those with the ambition to expand on this document are encouraged to contact the Computer Systems Research Group at ‘‘4bsd-ideas@Berkeley.Edu.’’

center box; cf s.

Groups of *dbx* Requests

_ l lfI lp-1fCW l. execution and tracing _ run execute object file cont continue execution from where it stopped trace display tracing information at specified place stop stop execution at specified place status display active *trace* and *stop* requests delete delete specific *trace* or *stop* requests catch start trapping specified signals ignore stop trapping specified signals step execute the next source line, stepping into functions next execute the next source line, even if it’s a function l lfI lp-1fCW l. _ displaying data _ print print the value of an expression whatis print the declaration of a given identifier or type which print outer block associated with identifier whereis print all symbols matching identifier assign set the value of a variable l lfI lp-1fCW l. _ function and procedure handling _ where display active procedures and functions on stack down move down the stack towards stopping point up move up the stack towards *main* call call the named function or procedure dump display names and values of all local variables l lfI lp-1fCW l. _ accessing source files and directories _ edit invoke an editor on current source file file change current source file func change the current function or procedure list display lines of source code use set directory list to search for source files /.../ search down in file to match regular expression ?...? search up in file to match regular expression l lfI lp-1fCW l. _ miscellaneous commands _ sh pass command line to the shell alias change *dbx* command name help explain commands source read commands from external file quit exit the debugger